CHAPTER 8

# Integration

This chapter covers different aspects of integration, focusing on numerical integration. For historical reasons, numerical integration is also known as *quadrature*. Integration is significantly more complex than its inverse operation—differentiation—and while many examples of integrals can be calculated analytically, in general, we must resort to numerical methods. Depending on the properties of the integrand (the function being integrated) and the integration limits, it can be easy or difficult to compute an integral numerically. In most cases, integrals of continuous functions and finite integration limits can be computed efficiently in one dimension. But integrable functions with singularities or integrals with infinite integration limits are cases that can be difficult to handle numerically, even in a single dimension. Two-dimensional integrals (double integrals) and higher-order integrals can be numerically computed with repeated single-dimension integration or using multidimensional generalizations of the techniques used to solve single-dimensional integrals. However, the computational complexity grows quickly with the number of dimensions to integrate. Such methods are only feasible for low-dimensional integrals like double or triple integrals. Integrals of higher dimensions than that often require entirely different techniques, such as Monte Carlo sampling algorithms.

In addition to the numerical evaluation of integrals with definite integration limits, which results in a single number, integration also has other important applications. For example, equations where the integrand of an integral is the unknown quantity are called integral equations, and such equations frequently appear in science and engineering applications. Integral equations are usually difficult to solve, but they can often be recast into linear equation systems by discretizing the integral. However, this topic is not covered here, but examples of this type of problem are shown in Chapter 10. Another important application of integration is integral transforms, which are techniques for transforming functions and equations between different domains. The end of this chapter briefly discusses how SymPy can compute some integral transforms, such as Laplace transforms and Fourier transforms.

To carry out symbolic integration, we can use SymPy, as briefly discussed in Chapter 3, and to compute numerical integration, the `integrate` module is mainly used in SciPy. However, SymPy (through the `mpmath` multiple-precision library) also has routines for numerical integration, which complement those in SciPy, for example, by offering arbitrary-precision integration. This chapter examines both options, discusses their pros and cons, and briefly looks at Monte Carlo integrations using the `scikit-monaco` library.

**scikit-monaco** `scikit-monaco` is a small library that makes Monte Carlo integration convenient and easily accessible. At the time of writing, the most recent version of scikit-monaco is 0.2.1. See `http://scikit-monaco.readthedocs.org` for more information.

## Importing Modules

This chapter requires, as usual, the NumPy and the Matplotlib libraries for basic numerical and plotting support, and on top of that, use the `integrate` module from SciPy, the SymPy library, and the `mpmath` arbitrary-precision math library. Here, let's assume that these modules are imported as follows.

```
In [1]: import numpy as np
In [2]: import matplotlib.pyplot as plt
   ...: import matplotlib as mpl
In [3]: from scipy import integrate
In [4]: import sympy
In [5]: import mpmath
```

In addition, for nicely formatted output from SymPy, we must set up its printing system.

```
In [6]: sympy.init_printing()
```

## Numerical Integration Methods

Here, we are concerned with evaluating definite integrals on the form $I(f)=\int_a^b f(x)\mathrm{d}x$, with given integration limits $a$ and $b$. The interval $[a, b]$ can be finite, semi-infinite (where either $a = -\infty$ or $b = \infty$), or infinite (where both $a = -\infty$ and $b = \infty$). The integral $I(f)$ can be interpreted as the area between the curve of the integrand $f(x)$ and the $x$ axis, as illustrated in Figure 8-1.

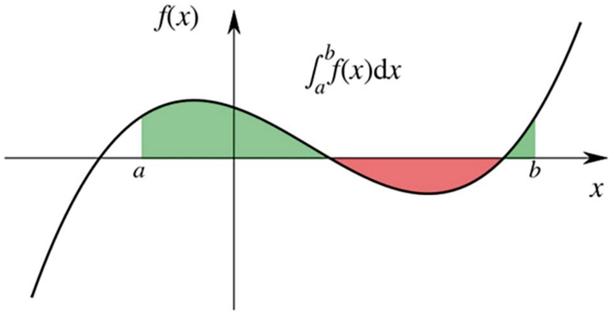

***Figure 8-1.*** *Interpretation of an integral as the area between the curve of the integrand and the x axis, where the area is counted as positive where f(x) > 0 (green/light) and negative otherwise (red/dark)*

A general strategy for numerically evaluating an integral $I(f)$, on the preceding form, is to write the integral as a discrete sum that approximates the value of the integral, as follows:

$$I(f)=\sum_{i=1}^{n} w_i f(x_i)+r_n.$$

Here, $w_i$ are the weights of $n$ evaluations of $f(x)$ at the points $x_i \in [a, b]$, and $r_n$ is the residual due to the approximation. In practice, assume that $r_n$ is small and can be neglected, but it is important to estimate $r_n$ to know how accurately the integral is approximated. This summation formula for $I(f)$ is known as an $n$-point *quadrature rule*, and the choice of the number of points $n$, their locations in $[a, b]$, and the weight factors $w_i$ influence the accuracy and the computational complexity of its evaluation. Quadrature rules can be derived from interpolations of $f(x)$ on the interval $[a, b]$. If the points $x_i$ are evenly spaced in the interval $[a, b]$, and a polynomial interpolation is used, then the resulting quadrature rule is known as a *Newton-Cotes quadrature rule*. For instance, approximating $f(x)$ with a zeroth-order polynomial (constant value) using the midpoint value $x_0 = (a+b)/2$, we obtain:

$$\int_a^b f(x)\mathrm{d}x \approx f\left(\frac{a+b}{2}\right)\int_a^b \mathrm{d}x = (b-a)f\left(\frac{a+b}{2}\right).$$

This is known as the *midpoint rule*, and it integrates polynomials of up to order one (linear functions) exactly, and it is therefore said to be of polynomial degree one. Approximating $f(x)$ by a polynomial of degree one, evaluated at the endpoints of the interval, results in the following:

$$\int_a^b f(x)\mathrm{d}x \approx \frac{b-a}{2}\left(f(a)+f(b)\right).$$

This is known as the *trapezoid rule and* is also of polynomial degree one. Using an interpolation polynomial of second order results in *Simpson's rule*,

$$\int_a^b f(x)\mathrm{d}x \approx \frac{b-a}{6}\left(f(a)+4f\left(\frac{a+b}{2}\right)+f(b)\right),$$

which uses function evaluations at the endpoints and the midpoint. This method is of polynomial degree three, meaning it integrates exactly polynomials up to order three. The method of arriving at this formula can easily be demonstrated using SymPy. First, define symbols for the $a$, $b$, and $x$ variables and the $f$ function.

```
In [7]: a, b, X = sympy.symbols("a, b, x")
In [8]: f = sympy.Function("f")
```

Next, define a tuple `x` that contains the sample points (the endpoints and the middle point of the interval $[a, b]$) and a list `w` of weight factors to be used in the quadrature rule, corresponding to each sample point.

```
In [9]: x = a, (a+b)/2, b  # for Simpson's rule
In [10]: w = [sympy.symbols("w_%d" % i) for i in range(len(x))]
```

Given `x` and `w`, we can now construct a symbolic expression for the quadrature rule.

```
In [11]: q_rule = sum([w[i] * f(x[i]) for i in range(len(x))])
In [12]: q_rule
```

Out[12]: $w_0 f(a) + w_1 f\left(\frac{a}{2}+\frac{b}{2}\right) + w_2 f(b)$

To compute the appropriate values of the weight factors $w_i$, choose the polynomial basis functions $\left\{\phi_n(x) = x^n\right\}_{n=0}^{2}$ for the interpolation of $f(x)$, and here, let's use the `sympy.Lambda` function to create symbolic representations for each of these basis functions.

```
In [13]: phi = [sympy.Lambda(X, X**n) for n in range(len(x))]
In [14]: phi
```

Out[14]: $[(x \mapsto 1), (x \mapsto x), (x \mapsto x^2)]$

The key to finding the weight factors in the quadrature expression (`Out[12]`) is that the integral $\int_a^b \phi_n(x)\mathrm{d}x$ can be computed analytically for each of the basis functions $\phi_n(x)$. By substituting the $f(x)$ function with each of the $\phi_n(x)$ basis functions in the quadrature rule, obtain an equation system for the unknown weight factors.

$$\sum_{i=0}^{2} w_i \phi_n(x_i) = \int_a^b \phi_n(x)\mathrm{d}x$$

These equations are equivalent to requiring that the quadrature rule exactly integrates all the basis functions and, therefore, (at least) all functions spanned by the basis. This equation system can be constructed with SymPy using the following.

```
In [15]: eqs = [q_rule.subs(f, phi[n]) - sympy.integrate(phi[n](X), (X, a, b))
    ...:        for n in range(len(phi))]
In [16]: eqs
```

Out[16]:
$$\left[a-b+w_0+w_1+w_2, \frac{a^2}{2}+aw_0-\frac{b^2}{2}+bw_2+w_1\left(\frac{a}{2}+\frac{b}{2}\right), \frac{a^3}{3}+a^2w_0--\frac{b^3}{3}+ b^2w_2+w_1\left(\frac{a}{2}+\frac{b}{2}\right)^2\right]$$

Solving this linear equation system gives analytical expressions for the weight factors.

```
In [17]: w_sol = sympy.solve(eqs, w)
In [18]: w_sol
```

Out[18]:
$$\left\{w_0: -\frac{a}{6}+\frac{b}{6}, w_1: -\frac{2a}{3}+\frac{2b}{3}, w_2: -\frac{a}{6}+\frac{b}{6}\right\}$$

Substituting the solution into the symbolic expression for the quadrature rule obtains the following.

```
In [19]: q_rule.subs(w_sol).simplify()
```

Out[19]:
$$-\frac{1}{6}(a-b)\left(f(a)+f(b)+4f\left(\frac{a}{2}+\frac{b}{2}\right)\right)$$

We recognize this result as Simpson's quadrature rule given in the preceding section. Choosing different sample points (the `x` tuple in this code) results in different quadrature rules.

Higher-order quadrature rules can similarly be derived using higher-order polynomial interpolation (more sample points in the $[a, b]$ interval). However, high-order polynomial interpolation can have undesirable behavior between the sample points, as discussed in Chapter 7. Rather than using higher-order quadrature rules, it is therefore often better to divide the integration interval $[a, b]$ into subintervals $[a = x_0, x_1], [x_1, x_2], \ldots, [x_{N-1}, x_N = b]$ and use a low-order quadrature rule in each of these subintervals. Such methods are known as *composite quadrature rules*. Figure 8-2 shows the three lowest-order Newton-Cotes quadrature rules for the $f(x) = 3+x+x^2+x^3+x^4$ function on the interval $[-1, 1]$ and the corresponding composite quadrature rules with four subdivisions of the original interval.

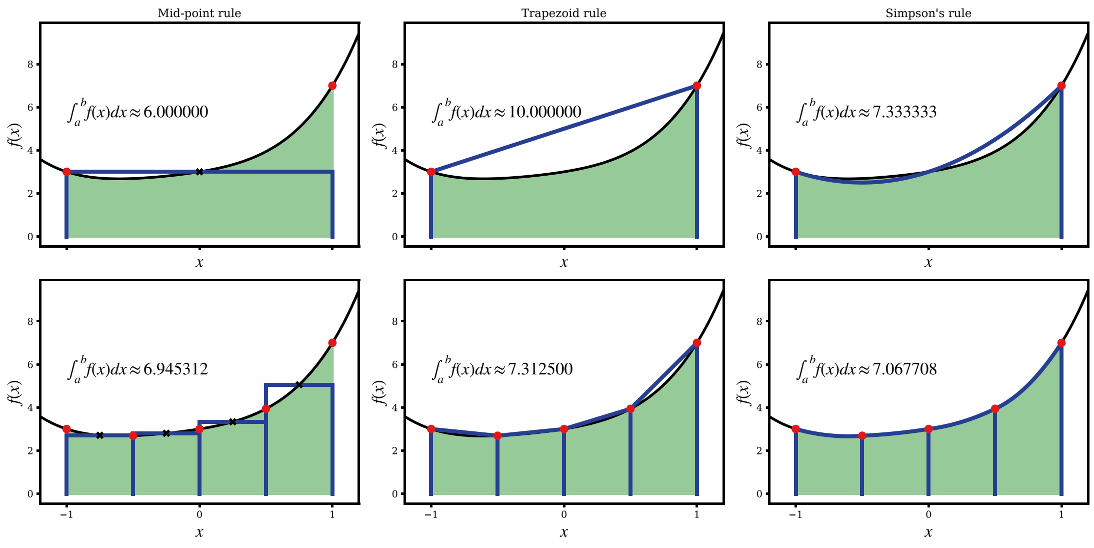

***Figure 8-2.*** *Visualization of quadrature rules (top panel) and composite quadrature rules (bottom panel) of orders zero (the midpoint rule), one (the trapezoid rule), and two (Simpson's rule)*

The subinterval length $h = (b - a)/N$ is an important parameter that characterizes composite quadrature rules. Estimates for the errors in an approximate quadrature rule and the scaling of the error with respect to $h$ can be obtained from Taylor series expansions of the integrand and the analytical integration of the term in the resulting series. An alternative technique simultaneously considers quadrature rules of different orders or subinterval lengths $h$. The difference between two such results can often be shown to give estimates of the error, and this is the basis for how many quadrature routines produce an estimate of the error in addition to the estimation of the integral, as shown in the examples in the following section.

We have seen that the Newton-Cotes quadrature rules use evenly spaced sample points of the integrand $f(x)$. This is often convenient, especially if the integrand is obtained from measurements or observations at prescribed points and cannot be evaluated at arbitrary points in the interval $[a, b]$. However, this is not necessarily the most efficient choice of quadrature nodes. If the integrand is given as a function that easily can be evaluated at arbitrary values of $x \in [a, b]$, then it can be advantageous to use quadrature rules that do not use evenly spaced sample points. An example of such a method is *Gaussian quadrature*, which also uses polynomial interpolation to determine the values of the weight factors in the quadrature rule but where the quadrature nodes $x_i$ are chosen to maximize the order of polynomials that can be integrated exactly (the polynomial degree) given a fixed number of quadrature points. It turns out that choices $x_i$ that satisfy these criteria are the roots of different orthogonal polynomials, and the sample points $x_i$ are typically located at irrational locations in the integration interval $[a, b]$. This generally is not a problem for numerical implementations. But practically, it requires that the $f(x)$ function is available to be evaluated at arbitrary points decided by the integration routine rather than given as tabulated or precomputed data at regularly spaced $x$ values. Gaussian quadrature rules are typically superior if $f(x)$ can be evaluated at arbitrary values. Yet, for the reason just mentioned, the Newton-Cotes quadrature rules also have important use cases when the integrand is given as tabulated data.

## Numerical Integration with SciPy

The numerical quadrature routines in the SciPy integrate module can be categorized into two types: those that take the integrand as a Python function and those that take arrays with samples of the integrand at given points. The functions of the first type use Gaussian quadrature (quad, quadrature, fixed_quad), while functions of the second type use Newton-Cotes methods (trapz, simps, and romb).

The quadrature function is an adaptive Gaussian quadrature routine implemented in Python. The quadrature repeatedly calls the fixed_quad function for Gaussian quadrature of a fixed order, increasing order until the required accuracy is reached. The quad function is a wrapper for routines from the *FORTRAN* library QUADPACK, which has superior speed performance and more features (such as support for infinite integration limits). It is, therefore, usually preferable to use quad, and the following uses this quadrature function. However, all these functions take similar arguments and can often be replaced with each other. They take as a first argument the function that implements the integrand, and the second and third arguments are the lower and upper integration limits. As a concrete example, consider the numerical evaluation of the integral $\int_{-1}^{1} e^{-x^2} \mathrm{d}x$. To evaluate this integral using SciPy's quad function, first define a function for the integrand and then call the quad function.

```
In [20]: def f(x):
    ...:     return np.exp(-x**2)
In [21]: val, err = integrate.quad(f, -1, 1)
In [22]: val
Out[22]: 1.493648265624854
In [23]: err
Out[23]: 1.6582826951881447e-14
```

The quad function returns a tuple that contains the numerical estimate of the integral, val, and an estimate of the absolute error, err. The tolerances for the absolute and the relative errors can be set using the optional epsabs and epsrel keyword arguments, respectively. If the f function takes more than one variable, the quad routine integrates the function over its first argument. We can optionally specify the values of additional arguments by passing those values to the integrand function via the keyword argument args to the quad function. For example, suppose we wish to evaluate $\int_{-1}^{1} a e^{-(x-b)^2/c^2} \mathrm{d}x$ for the specific values of the parameters $a = 1$, $b = 2$, and $c = 3$. We can define a function for the integrand that takes all these additional arguments and specifies the values of *a*, *b*, and *c* by passing args=(1, 2, 3) to the quad function.

```
In [24]: def f(x, a, b, c):
    ...:     return a * np.exp(-((x - b)/c)**2)
In [25]: val, err = integrate.quad(f, -1, 1, args=(1, 2, 3))
In [26]: val
Out[26]: 1.2763068351022229
In [27]: err
Out[27]: 1.4169852348169507e-14
```

When working with functions where the variable we want to integrate over is not the first argument, we can reshuffle the arguments by using a lambda function. For example, if we wish to compute the integral $\int_{0}^{5} J_0(x) \mathrm{d}x$, where the integrand $J_0(x)$ is the zeroth-order Bessel function of the first kind, it would be convenient to use the `jv` function from the `scipy.special` module as an integrand. The `jv` function takes the `v` and `x` arguments and is the Bessel function of the first kind for the real-valued order `v` and is evaluated at `x`. To use the `jv` function as an integrand for `quad`, we need to reshuffle the arguments of `jv`. With a lambda function, we can do this in the following manner.

```
In [28]: from scipy.special import jv
In [29]: f = lambda x: jv(0, x)
In [30]: val, err = integrate.quad(f, 0, 5)
In [31]: val
Out[31]: 0.7153119177847678
In [32]: err
Out[32]: 2.47260738289741e-14
```

With this technique, we can arbitrarily reshuffle arguments of any function and always obtain a function where the integration variable is the first argument so that it can be used as an integrand for `quad`.

The `quad` routine supports infinite integration limits. To represent infinite integration limits, we use the floating-point representation of infinity, `float('inf')`, which is conveniently available in NumPy as `np.inf`. For example, consider the integral $\int_{-\infty}^{\infty} e^{-x^2}\,\mathrm{d}x$. To evaluate it using `quad`, we can do

```
In [33]: f = lambda x: np.exp(-x**2)
In [34]: val, err = integrate.quad(f, -np.inf, np.inf)
In [35]: val
Out[35]: 1.7724538509055159
In [36]: err
Out[36]: 1.4202636780944923e-08
```

However, note that the `quadrature` and `fixed_quad` functions only support finite integration limits. With extra guidance, the `quad` function can also handle many integrals with integrable singularities. For example, consider the integral $\int_{-1}^{1} \frac{1}{\sqrt{|x|}}\,\mathrm{d}x$. The integrand diverges at $x = 0$, but the value of the integral does not diverge, and its value is 4. Naively trying to compute this integral using `quad` may fail because of the diverging integrand.

```
In [37]: f = lambda x: 1/np.sqrt(abs(x))
In [38]: a, b = -1, 1
In [39]: integrate.quad(f, a, b)
Out[39]: (NaN, NaN)
```

In situations like these, it can be helpful to graph the integrand to gain insights into its behavior, as shown in Figure 8-3.

```
In [40]: fig, ax = plt.subplots(figsize=(8, 3))
    ...: x = np.linspace(a, b, 10000)
    ...: ax.plot(x, f(x), lw=2)
    ...: ax.fill_between(x, f(x), color='green', alpha=0.5)
    ...: ax.set_xlabel("$x$", fontsize=18)
    ...: ax.set_ylabel("$f(x)$", fontsize=18)
    ...: ax.set_ylim(0, 25)
    ...: ax.set_xlim(-1, 1)
```

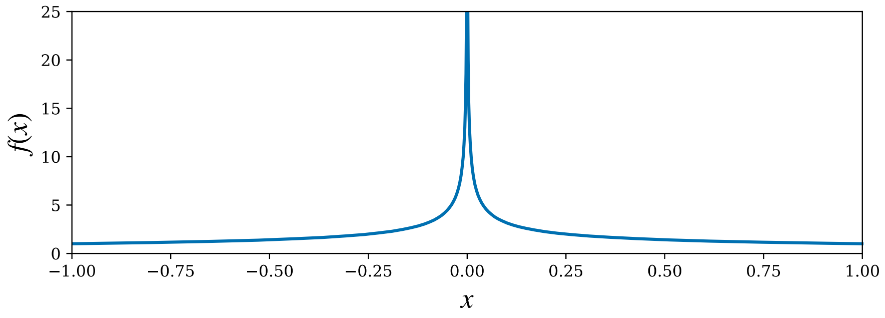

***Figure 8-3.*** *Example of a diverging integrand with finite integral (green/shaded area) that can be computed using the quad function*

In this case, the evaluation of the integral fails because the integrand diverges exactly at one of the sample points in the Gaussian quadrature rule (the midpoint). We can guide the quad routine by specifying a list of points that should be avoided using the `points` keyword arguments, and using `points=[0]` in the current example allows quad to evaluate the integral correctly.

```
In [41]: integrate.quad(f, a, b, points=[0])
Out[41]: (4.0,5.684341886080802e-14)
```

## Tabulated Integrand

We have seen that the quad routine is suitable for evaluating integrals when the integrand is specified using a Python function that the routine can evaluate at arbitrary points (determined by the specific quadrature rule). However, we may have an integrand that is only specified at predetermined points, such as evenly spaced points in the integration interval $[a, b]$. For example, this type of situation can occur when the integrand is obtained from experiments or observations that the particular integration routine cannot control. In this case, we can use a Newton-Cotes quadrature, such as the midpoint rule, trapezoid rule, or Simpson's rule described earlier in this chapter.

In the SciPy `integrate` module, the composite trapezoid rule and Simpson's rule are implemented in the `trapz` and `simps` functions. These functions take as first argument an array y with values of the integrand at a set of points in the integration interval, and they optionally take as second argument an array x that specifies the $x$ values of the sample points, or alternatively, the spacing dx between each sample (if uniform). Note that the sample points do not necessarily need to be evenly spaced, but they must be known in advance.

Let's evaluate an integral of a function that is given by sampled values using the integral $\int_0^2 \sqrt{x}\mathrm{d}x$ by taking 25 samples of the integrand in the integration interval [0, 2], as shown in Figure 8-4.

```
In [42]: f = lambda x: np.sqrt(x)
In [43]: a, b = 0, 2
In [44]: x = np.linspace(a, b, 25)
In [45]: y = f(x)
In [46]: fig, ax = plt.subplots(figsize=(8, 3))
    ...: ax.plot(x, y, 'bo')
    ...: xx = np.linspace(a, b, 500)
    ...: ax.plot(xx, f(xx), 'b-')
    ...: ax.fill_between(xx, f(xx), color='green', alpha=0.5)
    ...: ax.set_xlabel(r"$x$", fontsize=18)
    ...: ax.set_ylabel(r"$f(x)$", fontsize=18)
```

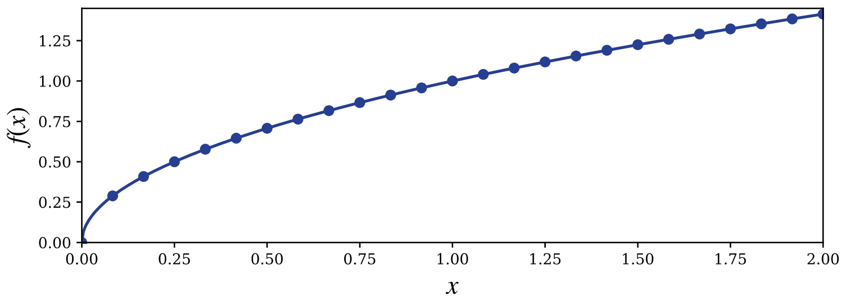

***Figure 8-4.** Integrand given as tabulated values marked with dots. The integral corresponds to the shaded area*

We can pass the x and y arrays to the `trapz` or methods to evaluate the integral. Note that the y array must be passed as the first argument.

```
In [47]: val_trapz = integrate.trapz(y, x)
In [48]: val_trapz
Out[48]: 1.88082171605
In [49]: val_simps = integrate.simps(y, x)
In [50]: val_simps
Out[50]: 1.88366510245
```

The `trapz` and `simps` functions do not provide any error estimates. For this example, we can compute the integral analytically and compare it to the numerical values computed with the two methods.

```
In [51]: val_exact = 2.0/3.0 * (b-a)**(3.0/2.0)
In [52]: val_exact
Out[52]: 1.8856180831641267
In [53]: val_exact - val_trapz
Out[53]: 0.00479636711328
In [54]: val_exact - val_simps
Out[54]: 0.00195298071541
```

Since all the information about the integrand is the given sample points, we cannot ask either `trapz` or `simps` to compute more accurate solutions. The only options for increasing the accuracy are expanding the number of sample points (which might be difficult if the underlying function is unknown) or possibly using a higherorder method.

The `integrate` module also implements the Romberg method with the `romb` function. The Romberg method is a Newton-Cotes method but one that uses Richardson extrapolation to accelerate the convergence of the trapezoid method; however, this method does require that the sample points are evenly spaced and also that there are $2^n+1$ sample points, where $n$ is an integer. Like the `trapz` and `simps` methods, `romb` takes an array with integrand samples as the first argument, but the second argument must (if given) be the sample-point spacing `dx`.

```
In [55]: x = np.linspace(a, b, 1 + 2**6)
In [56]: len(x)
Out[56]: 65
In [57]: y = f(x)
In [58]: dx = x[1] - x[0]
In [59]: val_exact - integrate.romb(y, dx=dx)
Out[59]: 0.000378798422913
```

Among the SciPy integration functions discussed here, `simps` is the most useful overall. It provides a good balance between ease of use (no constraints on the sample points) and relatively good accuracy.

## Multiple Integration

Multiple integrals, such as double integrals $\int_a^b\int_c^d f(x,y)\mathrm{d}x\mathrm{d}y$ and triple integrals $\int_a^b\int_c^d\int_e^f f(x,y,z)\mathrm{d}x\mathrm{d}y\mathrm{d}z$ , can be evaluated using the `dblquad` and `tplquad` functions from the SciPy integrate module. Also, integration over $n$ variables $\int\ldots\int_D f(\boldsymbol{x})\mathrm{d}\boldsymbol{x}$, over some domain $D$, can be evaluated using the `nquad` function. These functions are wrappers around the single-variable quadrature function `quad`, which is called repeatedly along each integral dimension.

Specifically, the double integral routine `dblquad` can evaluate integrals on the form

$$\int_a^b\int_{g(x)}^{h(x)} f(x,y)\mathrm{d}x\mathrm{d}y,$$

and it has the `dblquad(f, a, b, g, h)` function signature, where `f` is a Python function for the integrand, `a` and `b` are constant integration limits along the $x$ dimension, and `g` and `f` are Python functions (taking `x` as argument) that specify the integration limits along the $y$ dimension. For example, consider the integral $\int_0^1\int_0^1 e^{-x^2-y^2}\mathrm{d}x\mathrm{d}y$. To evaluate this, first define the `f` function for the integrand and graph the function and the integration region, as shown in Figure 8-5.

```
In [60]: def f(x, y):
    ...:     return np.exp(-x**2 - y**2)
In [61]: fig, ax = plt.subplots(figsize=(6, 5))
    ...: x = y = np.linspace(-1.25, 1.25, 75)
    ...: X, Y = np.meshgrid(x, y)
    ...: c = ax.contour(X, Y, f(X, Y), 15, cmap=mpl.cm.RdBu, vmin=-1, vmax=1)
    ...: bound_rect = plt.Rectangle((0, 0), 1, 1, facecolor="grey")
    ...: ax.add_patch(bound_rect)
    ...: ax.axis('tight')
    ...: ax.set_xlabel('$x$', fontsize=18)
    ...: ax.set_ylabel('$y$', fontsize=18)
```

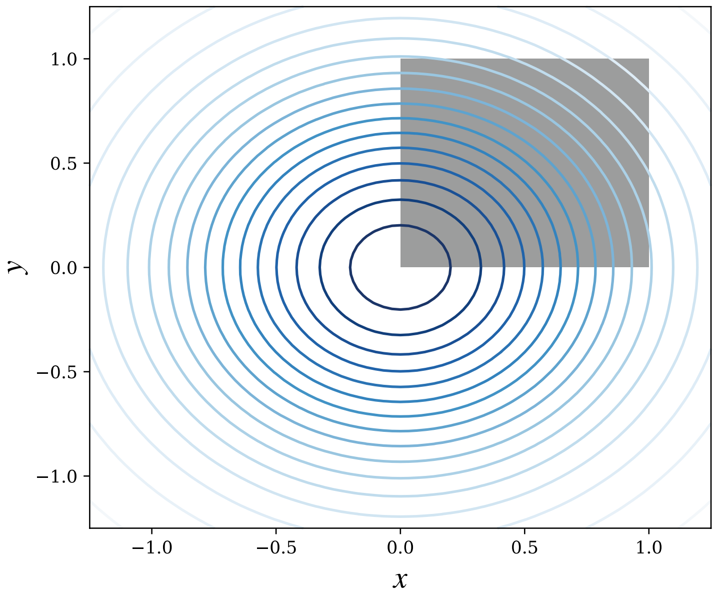

***Figure 8-5.*** *Two-dimensional integrand as a contour plot with integration region shown as a shaded area*

In this example, the integration limits for both the *x* and *y* variables are constants. But since `dblquad` expects functions for the integration limits for the *y* variable, we must also define the `h` and `g` functions, even though, in this case, they only evaluate to constants regardless of the value of `x`.

```
In [62]: a, b = 0, 1
In [63]: g = lambda x: 0
In [64]: h = lambda x: 1
```

Now, with all the arguments prepared, we can call `dblquad` to evaluate the integral.

```
In [65]: integrate.dblquad(f, a, b, g, h)
Out[65]: (0.5577462853510337, 6.1922276789587025e-15)
```

We could also do the same thing more concisely, although slightly less readable, by using inline `lambda` function definitions.

```
In [66]: integrate.dblquad(lambda x, y: np.exp(-x**2-y**2), 0, 1, lambda x: 0, lambda x: 1)
Out[66]: (0.5577462853510337, 6.1922276789587025e-15)
```

Because `g` and `h` are functions, we can compute integrals with *x*-dependent integration limits along the *y* dimension. For example, the following is obtained with $g(x) = x - 1$ and $h(x) = 1 - x$.

```
In [67]: integrate.dblquad(f, 0, 1, lambda x: -1 + x, lambda x: 1 - x)
Out[67]: (0.7320931000008094, 8.127866157901059e-15)
```

The tplquad function can compute integrals in the form

$$\int_a^b \int_{g(x)}^{h(x)} \int_{q(x,y)}^{r(x,y)} f(x,y,z)\mathrm{d}x\mathrm{d}y\mathrm{d}z,$$

which is a generalization of the double integral expression computed with dblquad. It additionally takes two Python functions as arguments that specify the integration limits along the z dimension. These functions take two arguments, x and y, but note that g and h only take one argument (x). To see how tplquad can be used, generalization of the previous integral to three variables: $\int_0^1\int_0^1\int_0^1 e^{-x^2-y^2-z^2}\mathrm{d}x\mathrm{d}y\mathrm{d}z$ . We compute this integral using a similar method compared to the dblquad example. We first define functions for the integrand and the integration limits and then call the tplquad function.

```
In [68]: def f(x, y, z):
    ...:    return np.exp(-x**2-y**2-z**2)
In [69]: a, b = 0, 1
In [70]: g, h = lambda x: 0, lambda x: 1
In [71]: q, r = lambda x, y: 0, lambda x, y: 1
In [72]: integrate.tplquad(f, 0, 1, g, h, q, r)
Out[72]: (0.4165383858866382, 4.624505066515441e-15)
```

For an arbitrary number of integrations, we can use the nquad function. It also takes the integrand as a Python function as the first argument. The integrand function should have the function signature f(x1, x2, ..., xn). In contrast to dplquad and tplquad, the nquad function expects a list of integration limit specifications as the second argument. The list should contain a tuple with integration limits for each integration variable or a callable function that returns such a limit. For example, we could use the following to compute the integral previously calculated with tplquad.

```
In [73]: integrate.nquad(f, [(0, 1), (0, 1), (0, 1)])
Out[73]: (0.4165383858866382, 8.291335287314424e-15)
```

For an increasing number of integration variables, the computational complexity of a multiple integral grows quickly, for example, when using nquad. To see this scaling trend, consider the following generalized version of the integrand studied with dplquad and tplquad.

```
In [74]: def f(*args):
    ...:    """
    ...:    f(x1, x2, ... , xn) = exp(-x1^2 - x2^2 - ... - xn^2)
    ...:    """
    ...:    return np.exp(-np.sum(np.array(args)**2))
```

Next, let's evaluate the integral for varying dimensions (ranging from 1 up to 5). In the following examples, the length of the list of integration limits determines the number of integrals. To see a rough estimate of the computation time, use the IPython %time command.

```
In [75]: %time integrate.nquad(f, [(0,1)] * 1)
CPU times: user 398 μs, sys: 63 μs, total: 461 μs
Wall time: 466 μs
Out[75]: (0.7468241328124271,8.291413475940725e-15)
In [76]: %time integrate.nquad(f, [(0,1)] * 2)
CPU times: user 6.31 ms, sys: 298 μs, total: 6.61 ms
```

```
Wall time: 6.57 ms
Out[76]: (0.5577462853510337,8.291374381535408e-15)
In [77]: %time integrate.nquad(f, [(0,1)] * 3)
CPU times: user 123 ms, sys: 2.46 ms, total: 126 ms
Wall time: 125 ms
Out[77]: (0.4165383858866382,8.291335287314424e-15)
In [78]: %time integrate.nquad(f, [(0,1)] * 4)
CPU times: user 2.41 s, sys: 11.1 ms, total: 2.42 s
Wall time: 2.42 s
Out[78]: (0.31108091882287664,8.291296193277774e-15)
In [79]: %time integrate.nquad(f, [(0,1)] * 5)
CPU times: user 49.5 s, sys: 169 ms, total: 49.7 s
Wall time: 49.7 s
Out[79]: (0.23232273743438786,8.29125709942545e-15)
```

Here, we see that increasing the number of integrations from one to five increases the computation time from hundreds of microseconds to nearly a minute. For an even larger number of integrals, it may become impractical to use direct quadrature routines, and other methods, such as Monte Carlo sampling techniques, can often be superior, especially if the required precision is low. Monte Carlo integration is a simple but powerful technique based on sampling the integrand at randomly selected points in the integral domain and gradually forming an estimate of the integral. Due to the stochastic nature of the algorithm, the conversion rate is typically relatively slow, and it is difficult to achieve very high accuracy. However, Monte Carlo integration scales very well with dimensionality and is often a competitive method for high-dimensional integrals.

To compute an integral using Monte Carlo sampling, we can use the mcquad function from the skmonaco library (known as scikit-monaco). As the first argument, it takes a Python function for the integrand; as the second argument, it takes a list of lower integration limits; and as the third argument, it takes a list of upper integration limits. Note that the way the integration limits are specified is different from for the quad function in SciPy's integrate module. Let's begin by importing the skmonaco (scikit-monaco) module.

```
In [80]: import skmonaco
```

Once the module is imported, we can use the skmonaco.mcquad function for performing a Monte Carlo integration. The following example computes the same integral as in the previous example using nquad.

```
In [81]: %time val, err = skmonaco.mcquad(f, xl=np.zeros(5), xu=np.ones(5), npoints=100000)
CPU times: user 1.43 s, sys: 100 ms, total: 1.53 s
Wall time: 1.5 s
In [82]: val, err
Out[82]: (0.231322502809, 0.000475071311272)
```

While the error is not comparable to the result given by nquad, the computation time is much shorter. By increasing the number of sample points, which we can specify using the npoints argument, we can increase the accuracy of the result. However, the convergence of Monte Carlo integration is very slow, and it is most suitable when high accuracy is not required. However, the beauty of Monte Carlo integration is that its computational complexity is independent of the number of integrals. This is illustrated in the following example, which computes a ten-variable integration at the same time and with a comparable error level as the previous example with a five-variable integration.

```
In [83]: %time val, err = skmonaco.mcquad(f, xl=np.zeros(10), xu=np.ones(10),
npoints=100000)
CPU times: user 1.41 s, sys: 64.9 ms, total: 1.47 s
```

```
Wall time: 1.46 s
In [84]: val, err
Out[84]: (0.0540635928549, 0.000171155166006)
```

## Symbolic and Arbitrary-Precision Integration

Chapter 3 presented examples of how SymPy can compute definite and indefinite integrals of symbolic functions using the `sympy.integrate` function. For example, to compute the integral $\int_{-1}^{1} 2\sqrt{1-x^2}\,\mathrm{d}x$ first create a symbol for $x$ and define expressions for the integrand, and the integration limits $a = -1$ and $b = 1$.

```
In [85]: x = sympy.symbols("x")
In [86]: f = 2 * sympy.sqrt(1-x**2)
In [87]: a, b = -1, 1
```

Next, we can compute the closed-form expression for the integral using the following.

```
In [88]: val_sym = sympy.integrate(f, (x, a, b))
In [89]: val_sym
Out[89]: π
```

For this example, SymPy can find the analytic expression for the integral, $\pi$. As pointed out earlier, this situation is the exception, and in general, we cannot find an analytical closed-form expression. We then need to resort to numerical quadrature, for example, using SciPy's `integrate.quad`, as discussed earlier in this chapter. However, the `mpmath` library,[1] which is closely integrated with SymPy, provides an alternative implementation of numerical quadrature using arbitrary-precision computations. With this library, we can evaluate an integral to arbitrary precision without being restricted by the limitations of floating-point numbers. However, the downside is, of course, that arbitrary-precision computations are significantly slower than floating-point computations. But when we require precision beyond what the SciPy quadrature functions can provide, this multiple-precision quadrature offers a solution.

For example, to evaluate the integral $\int_{-1}^{1} 2\sqrt{1-x^2}\,\mathrm{d}x$ to a given precision,[2] we can use the `mpmath.quad` function, which takes a Python function for the integrand as the first argument and the integration limits as a tuple `(a, b)` as the second argument. To specify the precision, set the variable `mpmath.mp.dps` to the required number of accurate decimal places. For example, if we need 75 accurate decimal places, set the following.

```
In [90]: mpmath.mp.dps = 75
```

The integrand must be given as a Python function that uses math functions from the `mpmath` library to compute the integrand. From a SymPy expression, we can create such a function using `sympy.lambdify` with `'mpmath'` as the third argument, which indicates that we want an `mpmath` compatible function. Alternatively,

[1] For more information about the multiprecision (arbitrary precision) math library mpmath, see the project's web page at http://mpmath.org.

[2] Here, I deliberately choose to work with an integral with a known analytical value to compare the multiprecision quadrature result with the known exact value.

we can directly implement a Python function using the math functions from the `mpmath` module in SymPy, which would be `f_mpmath = lambda x: 2 * mpmath.sqrt(1 - x**2)`. However, let's use `sympy.lambdify` to automate this step.

```
In [91]: f_mpmath = sympy.lambdify(x, f, 'mpmath')
```

Next, compute the integral using `mpmath.quad` and display the resulting value.

```
In [92]: val = mpmath.quad(f_mpmath, (a, b))
In [93]: sympy.sympify(val)
Out[93]: 3.14159265358979323846264338327950288419716939937510582097494459230781640629
```

To verify that the numerically computed value is accurate to the required number of decimal places (75), compare the result with the known analytical value ($\pi$). The error is indeed very small.

```
In [94]: sympy.N(val_sym, mpmath.mp.dps+1) - val
Out[94]: 6.90893484407555570030908149024031965689280029154902510801896277613487344253e-77
```

This level of precision cannot be achieved with the quad function in SciPy's `integrate` module since the precision of floating-point numbers limits it.

The `mpmath` library's `quad` function can also evaluate double and triple integrals. To do so, we only need to pass to it an integrand function that takes multiple variables as arguments and pass tuples with integration limits for each integration variable. For example, to compute the double integral

$$\int_0^1\int_0^1 \cos(x)\cos(y)e^{-x^2-y^2}\mathrm{d}x\mathrm{d}y$$

and the triple integral

$$\int_0^1\int_0^1\int_0^1 \cos(x)\cos(y)\cos(z)e^{-x^2-y^2-z^2}\mathrm{d}x\mathrm{d}y\mathrm{d}z$$

to 30 significant decimals (this example cannot be solved symbolically with SymPy), we could first create SymPy expressions for the integrands, and then use `sympy.lambdify` to create the corresponding `mpmath` expressions.

```
In [95]: x, y, z = sympy.symbols("x, y, z")
In [96]: f2 = sympy.cos(x) * sympy.cos(y) * sympy.exp(-x**2 - y**2)
In [97]: f3 = sympy.cos(x) * sympy.cos(y) * sympy.cos(z) * sympy.exp(-x**2 - y**2 - z**2)
In [98]: f2_mpmath = sympy.lambdify((x, y), f2, 'mpmath')
In [99]: f3_mpmath = sympy.lambdify((x, y, z), f3, 'mpmath')
```

The integrals can then be evaluated to the desired accuracy by setting `mpmath.mp.dps` and calling `mpmath.quad`.

```
In [100]: mpmath.mp.dps = 30
In [101]: mpmath.quad(f2_mpmath, (0, 1), (0, 1))
Out[101]: mpf('0.430564794306099099242308990195783')
In [102]: res = mpmath.quad(f3_mpmath, (0, 1), (0, 1), (0, 1))
In [103]: sympy.sympify(res)
Out[103]: 0.282525579518426896867622772405
```

Again, this gives access to accuracy levels beyond what `scipy.integrate.quad` can achieve, but this additional accuracy comes with a hefty increase in computational cost. Note that the type of the object returned by `mpmath.quad` is a multiprecision float (`mpf`). It can be cast into a SymPy type using `sympy.sympify`.

## Line Integrals

SymPy can also compute line integrals on the form $\int_C f(x, y)\mathrm{d}s$, where $C$ is a curve in the $x$–$y$ plane, using the `line_integral` function. This function takes the integrand, as a SymPy expression, as the first argument, a `sympy.Curve` instance as the second argument, and a list of integration variables as the third argument. The path of the line integral is specified by the `Curve` instance, which describes a parameterized curve for which the $x$ and $y$ coordinates are given as a function of an independent parameter, $t$. We can use the following to create a Curve instance that describes a path along the unit circle.

```
In [104]: t, x, y = sympy.symbols("t, x, y")
In [105]: C = sympy.Curve([sympy.cos(t), sympy.sin(t)], (t, 0, 2 * sympy.pi))
```

Once the integration path is specified, we can easily compute the corresponding line integral for a given integrand using `line_integral`. For example, with the integrand $f(x, y) = 1$, the result is the circumference of the unit circle.

```
In [106]: sympy.line_integrate(1, C, [x, y])
Out[106]: 2π
```

The result is less obvious for a nontrivial integrand, such as in the following example, which computes the line integral with the integrand $f(x, y) = x^2y^2$.

```
In [107]: sympy.line_integrate(x**2 * y**2, C, [x, y])
Out[107]: π/4
```

## Integral Transforms

The last application of integrals discussed in this chapter is integral transforms. An integral transform is a procedure that takes a function as input and outputs another function. Integral transforms are the most useful when they can be computed symbolically, and here we explore two examples of integral transforms that can be performed using SymPy: the Laplace transform and the Fourier transform. These two transformations have numerous applications, but the fundamental motivation is to transform problems into a more easily handled form. It can, for example, be a transformation of a differential equation into an algebraic equation, using Laplace transforms, or a transformation of a problem from the time domain to the frequency domain using Fourier transforms.

In general, an integral transform of the $f(t)$ function can be written as

$$T_f(u) = \int_{t_1}^{t_2} K(t,u) f(t)\mathrm{d}t,$$

where $T_f(u)$ is the transformed function. The choice of the kernel $K(t, u)$ and the integration limits determine the type of integral transform.

The inverse of the integral transform is given by

$$f(u) = \int_{u_1}^{u_2} K^{-1}(u,t) T_f(u) \mathrm{d}u,$$

where $K^{-1}(u, t)$ is the kernel of the inverse transform. SymPy provides functions for several types of integral transform.

Let's focus on the Laplace transform

$$L_f(s) = \int_0^\infty e^{-st} f(t) \mathrm{d}t,$$

with the inverse transform

$$f(t) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} e^{st} L_f(s) \mathrm{d}s.$$

Let's also focus on the Fourier transform

$$F_f(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-i\omega t} f(t) \mathrm{d}t,$$

with the inverse transform

$$f(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{i\omega t} F_f(\omega) d\omega.$$

With SymPy, we can perform these transforms with the `sympy.laplace_transform` and `sympy.fourier_transform`, respectively, and the corresponding inverse transforms can be computed with the `sympy.inverse_laplace_transform` and `sympy.inverse_fourier_transform`. These functions take a SymPy expression for the function to transform as the first argument and the symbol for the independent variable of the expression to transform as the second argument (e.g., $t$). As the third argument they take the symbol for the transformation variable (e.g., $s$). For example, to compute the Laplace transform of the $f(t) = sin\,(at)$ function, we begin by defining SymPy symbols for the variables $a$, $t$, and $s$ and a SymPy expression for the $f(t)$ function.

```
In [108]: s = sympy.symbols("s")
In [109]: a, t = sympy.symbols("a, t", positive=True)
In [110]: f = sympy.sin(a*t)
```

Once we have SymPy objects for the variables and the function, we can call the `laplace_transform` function to compute the Laplace transform.

```
In [111]: sympy.laplace_transform(f, t, s)
```

Out[111]: $\left(\frac{a}{a^2+s^2}, -\infty, 0 < \Re s\right)$

By default, the `laplace_transform` function returns a tuple containing the resulting transform; the value $A$ from the convergence condition of the transform, which takes the form $A < \Re s$; and lastly, additional conditions required for the transform to be well defined. These conditions typically depend on the constraints specified when symbols are created. For example, positive=True was used when creating the symbols *a* and *t* to indicate that they represent real and positive numbers. Often, we are only interested in the transform itself, and we can then use the `noconds=True` keyword argument to suppress the conditions in the return result.

```
In [112]: F = sympy.laplace_transform(f, t, s, noconds=True)
In [113]: F
```

Out[113]: $\frac{a}{a^2 + s^2}$

The inverse transformation can be used similarly, except we need to reverse the roles of the symbols `s` and `t`. The Laplace transform is a unique one-to-one mapping, so if we compute the inverse Laplace transform of the previously computed Laplace transform, we expect to recover the original function.

```
In [114]: sympy.inverse_laplace_transform(F, s, t, noconds=True)
Out[114]: sin(at)
```

SymPy can compute the transforms for many elementary mathematical functions and combinations of such functions. When solving problems using Laplace transformations by hand, one typically searches for matching functions in reference tables with known Laplace transformations. Using SymPy, this process can conveniently be automated in many, but not all, cases. The following examples show a few additional examples of well-known functions that one finds in Laplace transformation tables. Polynomials have simple Laplace transformation.

```
In [115]: [sympy.laplace_transform(f, t, s, noconds=True) for f in [t, t**2, t**3, t**4]]
```

Out[115]: $[\frac{1}{s^2}, \frac{2}{s^3}, \frac{6}{s^4}, \frac{24}{s^5}]$

We can also compute the general result with an arbitrary integer exponent.

```
In [116]: n = sympy.symbols("n", integer=True, positive=True)
In [117]: sympy.laplace_transform(t**n, t, s, noconds=True)
```

Out[117]: $\frac{\Gamma(n+1)}{s^{n+1}}$

The Laplace transform of composite expressions can also be computed, as in the following example, which computes the transform of the $f(t) = (1 - at)e^{-at}$ function.

```
In [118]: sympy.laplace_transform((1 - a*t) * sympy.exp(-a*t), t, s, noconds=True)
```

Out[118]: $\frac{s}{(a+s)^2}$

The main application of Laplace transforms is to solve differential equations, where the transformation can be used to bring the differential equation into a purely algebraic form, which can then be solved and transformed back to the original domain by applying the inverse Laplace transform. Chapter 9 provides concrete examples of this method. Fourier transforms can also be used for the same purpose.

The Fourier transform function, `fourier_tranform`, and its inverse, `inverse_fourier_transform`, are used in much the same way as the Laplace transformation functions. For example, to compute the Fourier transform $f(t) = e^{-at^2}$, we would first define SymPy symbols for the variables $a$, $t$, and $\omega$, and the $f(t)$ function, then compute the Fourier transform by calling the `sympy.fourier_tranform` function.

```
In [119]: a, t, w = sympy.symbols("a, t, omega")
In [120]: f = sympy.exp(-a*t**2)
In [121]: F = sympy.fourier_transform(f, t, w)
In [122]: F
```

Out[122]: $\sqrt{\pi / a} e^{-\pi^2 \omega^2 / a}$

As expected, computing the inverse transformation for `F` recovers the original function.

```
In [123]: sympy.inverse_fourier_transform(F, w, t)
```

Out[123]: $e^{-at^2}$

SymPy can be used to compute a wide range of Fourier transforms symbolically, and it can be a valuable tool for solving time-dependent equations in the frequency domain. Unfortunately, SymPy does not yet handle transformations involving Dirac delta functions well, either in the original or the resulting transformation. This currently limits its usability in many applications where the resulting equations contain single-frequency components.

## Summary

Integration is one of the fundamental tools in mathematical analysis. Numerical quadrature, or numerical evaluation of integrals, has important applications in many fields of science because integrals that occur in practice often cannot be computed analytically and expressed as a closed-form expression. Their computation then requires numerical techniques. This chapter reviewed basic techniques and methods for numerical quadrature and introduced the corresponding functions in the SciPy `integrate` module that can be used to evaluate integrals in practice. When the integrand is given as a function that can be evaluated at arbitrary points, we typically prefer Gaussian quadrature rules.

On the other hand, when the integrand is defined as tabulated data, the more straightforward Newton-Cotes quadrature rules can be used. We also studied symbolic integration and arbitrary-precision quadrature, which can complement floating-point quadrature for specific integrals that can be computed symbolically or when additional precision is required. As usual, a good starting point is to analyze a problem symbolically, and if a particular integral can be solved symbolically by finding its antiderivative, that is generally the most desirable outcome. When symbolic integration fails, we must resort to numerical quadrature, which should first be explored with floating-point-based implementations, like the ones provided by the SciPy `integrate` module. If additional accuracy is required, we can fall back on arbitrary-precision quadrature. Another application of symbolic integration is integral transformation, which can be used to transform problems, such as differential equations, between different domains. This chapter briefly examined how to perform Laplace and Fourier transforms symbolically using SymPy. The next chapter continues to explore this for solving certain types of differential equations.

## Further Reading

Numerical quadrature is discussed in many introductory textbooks on numerical computing, such as *Scientific Computing: An Introductory Survey* by M. T. Heath (McGraw-Hill, 2002) and *Introduction to Numerical Analysis* by J. Stoer et al. (Springer, 1992). Quadrature methods and implementations are discussed in *Numerical Recipes in C* by W. H. Press (Cambridge University Press, 2002). For the theory of integral transforms, such as the Fourier and the Laplace transforms, see *Fourier Analysis and Its Applications* by G. B. Folland (American Mathematical Society, 1992).